Resilient State Machine Replication

Paulo Sousa
Nuno Ferreira Neves
Paulo Veríssimo

DI–FCUL TR–05–17

September 2005

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1749–016 Lisboa
Portugal

Technical reports are available at http://www.di.fc.ul.pt/tech-reports. The files are stored in PDF, with the report number as filename. Alternatively, reports are available by post from the above address.

Resilient State Machine Replication*

Paulo Sousa, Nuno Ferreira Neves and Paulo Veríssimo

University of Lisboa, Portugal
{pjsousa, nuno, pjv}@di.fc.ul.pt

Abstract

Nowadays, one of the major concerns about the services provided over the Internet is related to their availability. Replication is a well known way to increase the availability of a service. However, replication has some associated costs, namely it is necessary to guarantee a correct coordination between the replicas. Moreover, being the Internet such an unpredictable and insecure environment, coordination correctness should be tolerant to Byzantine faults and immune to timing failures. Several past works address agreement and replication techniques that tolerate Byzantine faults under the asynchronous model, but they all make the assumption that the number of faulty replicas is bounded and known. Assuming a maximum number of f faulty replicas under the asynchronous model is dangerous – there is no way of guaranteeing that no more than f faults will occur during the execution of the system. In this paper, we propose a new design methodology, in order to build a resilient f fault/intrusion-tolerant state machine replication system, which guarantees that no more than f faults ever occur. The system is asynchronous in its most part and it resorts to a synchronous oracle to periodically remove the effects of faults/attacks from the replicas.

1 Introduction

Nowadays, one of the major concerns about the services provided by computer systems is related to their availability. This applies specially to services provided over the Internet. Building highly available services involves, on one hand, the design and implementation of correct services tolerant to Byzantine faults [21, 14], and on other hand, the assurance that the access to them is always guaranteed with an high probability. Interestingly, these two tasks can be both accomplished by recurring to replication techniques.

Replication is a well known way to improve the availability of a service: if a service can be accessed through different independent paths, then the probability

*This work was partially supported by the FCT, through the Large-Scale Informatic Systems Laboratory (LaSIGE).

of a client being able to use it increases. But replication has costs, namely it is necessary to guarantee a correct coordination between the replicas. Moreover, the Internet being an unpredictable and insecure environment, coordination correctness should be assured under the worst possible operation conditions: absence of (local or distributed) timing guarantees and possibility of Byzantine faults triggered by malicious adversaries.

Several past works address agreement and replication techniques that tolerate Byzantine faults under the asynchronous model. The majority of these techniques make the assumption that the number of faulty replicas is bounded by a known value [2, 5, 17, 16, 9, 18, 4, 7].

This assumption is of course non-controversial in the context of an abstract algorithm design, but care should be taken when the same algorithm is used to implement a system. Systems are supposed to work in the real world, under actual physical constraints and concrete assumptions. So, the system assumption of 'known maximum bound on the number of failures' deserves a closer look. In fact, under the asynchronous model, this type of assumption may be dangerous. There is no way of guaranteeing that no more than f faults will occur during the execution of the system offering the service: unbounded processing and message delays may result in a very long execution time.

Recent works [6, 27, 3, 19] use the proactive recovery approach [20] with the goal of weakening the assumption on the number of faults. The above mentioned assumption 'known maximum bound on the number of failures' is confined to a window of vulnerability, which in turn would allow the algorithms proposed in these papers to tolerate any number of faults over the lifetime of the system. However, this window is again defined under the asynchronous model, and can therefore have an unbounded length. This is specially true in a malicious environment, such as the Internet.

In a recent work, we looked at this problem with the help of a novel theoretical Physical System Model (PSM) [24]. PSM takes in account the environmental resources and their evolution along the timeline of system execution. The model builds on the concept of *resource exhaustion* – the situation when a system no longer has the necessary resources to execute correctly (e.g., bandwidth, replicas). PSM allowed us to introduce the predicate *exhaustion-safe*, meaning freedom from *exhaustion-failures* – failures that result from accidental or provoked resource exhaustion. We showed that, under the asynchronous model, it is theoretically impossible to have an exhaustion-safe replication technique that can only tolerate a bounded number of faults, even if we enhance it with proactive recovery [25].

This theoretical result applies to most of the practical systems deployed over the Internet, specially to those using replication to achieve fault-tolerance and availability. Practical systems are typically not completely asynchronous under normal operation – some eventual guarantees can be given on the bounds of processing and message delays. However, in an environment prone to malicious faults an adversary may make the system as asynchronous as she or he wants.

Therefore, a replicated system is adequate to be deployed in an asynchronous

and insecure environment, such as the Internet, if it does not make timing assumptions and if it does not assume a maximum number of faulty replicas. This could be done by enhancing the system with a detection mechanism responsible for detecting faulty replicas and recovering them. This is relatively easy if replicas can only suffer crash or omission faults, but things get more complicated with Byzantine faults – a malicious adversary may remain dormant until the compromise of $f + 1$ replicas and only deploy the "real" attack afterwards. Thus, it is complex to build a reliable fault detection mechanism able to detect arbitrary faults.

Given that it is difficult to detect faults, and assuming that compromising a replica takes some time, in alternative one can calculate the minimum time necessary for $f + 1$ replicas to be compromised and periodically trigger the execution of a recovering procedure. If an appropriate triggering period is chosen and if the recovering procedure is timely executed in every replica, one can guarantee that no more than f faults will occur, for some f. In a recent work we present a system design methodology – $\mathcal{M}_{exhaustion-safe}$ – based on this reasoning and formally prove that it allows the construction of exhaustion-safe systems [25]. This design methodology uses proactive resilience, which is a new approach to proactive recovery based on architectural hybridization.

In this paper, we propose to use the design methodology $\mathcal{M}_{exhaustion-safe}$ in order to build an exhaustion-safe, and thus resilient, state machine replication system. A rejuvenation protocol is presented and the conditions for exhaustion-safety are derived. The protocol is executed by a synchronous and secure component, which guarantees that these conditions are either satisfied or the system switches to a fail-safe state.

The paper is organized as follows. Section 2 extends the concept of exhaustion-safety presented in [24] and revisits the design methodology $\mathcal{M}_{exhaustion-safe}$ presented in [25]. In Section 3, we explain how to build an exhaustion-safe state machine replication system. Related work is discussed in Section 4 and, finally, our conclusions and future work are presented in Section 5.

2 Exhaustion-Safety Revisited

In this section we first revise and extend the concept of exhaustion-safety introduced in [24], and then explain the generic design methodology presented in [25] that should be applied in order to build exhaustion-safe systems. These notions are fundamental for the understanding of the results of this paper.

2.1 Exhaustion-Safety Definition

Typically, the correctness of a protocol depends on a set of assumptions regarding aspects like the processing power, the type and number of faults that can happen, the synchrony of the execution, etc. These assumptions are in fact an abstraction of the actual resources the protocol needs to work correctly (e.g., when a protocol

assumes that messages are delivered within a known bound, it is in fact assuming that the network will have certain characteristics such as bandwidth and latency). The violation of these resource assumptions may affect the safety and/or liveness of the protocol. If the protocol is vital for the operation of some system, then the system liveness and/or safety may also be affected.

We classify resource assumptions as *safety resource assumptions* and/or *liveness resource assumptions*, depending on whether they affect, respectively, the safety and/or liveness of the system (by affecting the safety/liveness of some vital protocol). Additionally, we distinguish between *quantitative* resource assumptions and *non-quantitative* resource assumptions. The former specify a resource threshold (e.g., a channel loses at most f consecutive messages, a replicated system tolerates at most f replica failures), and the latter a non-quantitative resource behaviour (e.g., a channel is reliable, a channel guarantees total order, a replicated system tolerates Byzantine failures).

We are specially interested on safety resource assumptions, given that the correctness of a system depends on them. More precisely, our focus is on quantitative safety resource assumptions. Resource exhaustion is defined in the following manner:

Definition 2.1. *A resource r is said to be exhausted if there exists a quantitative safety assumption on r and this assumption is violated.*

It is impossible to guarantee the correctness of a system with one or more exhausted resources. Therefore, exhaustion-safety is defined in the following manner.

Definition 2.2. *Exhaustion-safety with regard to a resource r is the ability of a system to ensure that r is not exhausted.*

Consequently, an exhaustion-safe system is defined in the following way.

Definition 2.3. *A system is r-exhaustion-safe if it satisfies the exhaustion-safety property with regard to resource r.*

We argue that a system, namely a distributed system, in order to be dependable, has to satisfy the exhaustion-safety property with regard to all the resources.

2.2 Asynchrony and Replica-Exhaustion-Safety

Consider a distributed f fault-tolerant replicated system. According to Definition 2.3, this system is *replica*-exhaustion-safe if it guarantees that no quantitative safety assumption on *replicas* is violated – this means that a maximum of f replica failures may occur during system execution. In order to guarantee that no more than f replicas fail, one has to guarantee that the system terminates its execution before the time needed for $f + 1$ failures to be produced. In practical terms, one would like to foresee the maximum number of failures bound to occur during the system execution, call it N_f, so that it is designed to tolerate $f \geq N_f$ failures.

It is easy to see that estimating N_f is rather difficult in systems built under the asynchronous model: the absence of timing assumptions dictates unbounded drift rates for the local clocks, arbitrary delays for local processing, and message delivery [10, 15, 8]. The execution time of a system built under the asynchronous model is unbounded, and thus the estimation of N_f becomes theoretically impossible. Therefore, one can say (and we formally proved this in [24]) that a distributed f fault-tolerant replicated system built under the asynchronous model is not replica-exhaustion-safe.

However, the asynchronous model is quite attractive because it leads to the design of programs and components that are immune to timing failures. This characteristic is very important if we think of systems deployed over environments subject to unpredictable delays, such as the Internet. Consequently, the ideal replicated system would be a replica-exhaustion-safe one which could somehow maintain the nice characteristics of the asynchronous model.

2.3 Towards Replica-Exhaustion-Safety

In a recent work, we propose to circumvent the impossibility result presented in the previous section, through the Proactive Resilience Model (PRM) [25] – a new approach to proactive recovery [20] based on architectural hybridization [26]. The idea is that given a distributed f fault-tolerant replicated system A built under the asynchronous model, one can make it replica-exhaustion-safe, by redesigning A as being part of an architecturally hybrid distributed system A^*, composed by A and by a synchronous and secure proactive recovery subsystem R. R periodically and timely rejuvenates A, so that it is impossible for an adversary to provoke more than f replica failures. The resulting system A^* is partially synchronous, given that it is composed by the asynchronous system A and by the synchronous system R.

In [25] the proactive recovery subsystem R is modelled as an abstract component, the Proactive Recovery Wormhole (PRW), and we present a design methodology – $\mathcal{M}_{exhaustion-safe}$ – which uses the PRW to build replica-exhaustion-safe distributed f fault-tolerant replicated systems. We now describe these in detail.

2.3.1 The Proactive Recovery Wormhole

The Proactive Recovery Wormhole (PRW) is an abstract secure real-time distributed component that aims to execute proactive recovery procedures. By abstract we mean that the PRW allows many instantiations. Typically, an instantiation is chosen according to the concrete application/protocol that needs to be proactively recovered.

The architecture of a system with a PRW is suggested in Figure 1. An architecture with a PRW has a local module in some hosts, called the *local PRW*. Depending on the instantiation, these modules may or may not be interconnected by a *control network*. This set up of local PRW modules optionally interconnected by the control network is collectively called *the* PRW. The PRW is used to execute

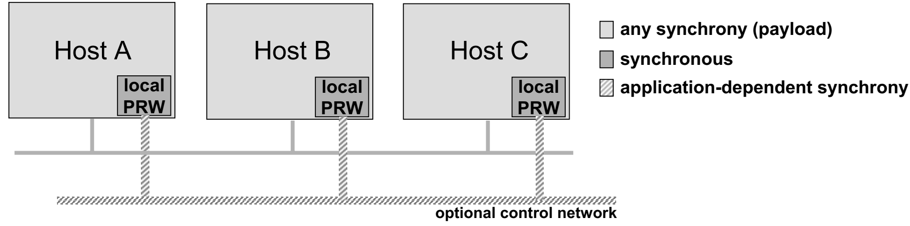

Figure 1: The architecture of a system with a PRW.

proactive recovery procedures of protocols/applications running between participants in the hosts concerned, on any usual distributed system architecture (e.g., the Internet). We call the latter the *payload system and network*, to differentiate from the PRW part.

Conceptually, a local PRW should be considered to be a module inside a host, and separated from the OS. In practice, this conceptual separation between the local PRW and the OS can be achieved in several ways: (1) the local PRW can be implemented in a separate, tamper-proof hardware module (e.g., PC board) and so the separation is physical; (2) the local PRW can be implemented on the native hardware, with a virtual separation and shielding implemented in software, between the former and the OS processes.

The local PRWs are assumed to be fail-silent (they fail by crashing). Every local PRW preserves, by construction, the following properties:

P1 There exists a known upper bound $T_{proc_{max}}$ on the processing delays;

P2 There exists a known upper bound $T_{drift_{max}}$ on the drift rate of local clocks.

As mentioned, a PRW instantiation may or may not have a control network. For instance, if a proactive recovery procedure only requires local information, then the control network is expendable. Even when the control network is required, its characteristics will depend on the specific requirements of the proactive recovery procedure.

The PRW offers a single service: *periodic timely execution*. This service can be defined as follows:

Definition 2.4. *Given any function F, with a calculated worst case execution time of T_{Xmax}, an execution interval T_d, and a period T_p, satisfying $T_{Xmax} < T_d < T_p$ (see Figure 2), then F is triggered by the PRW* ***periodic timely execution service*** *at real time instants t_i (the i-th triggering occurs at instant t_i), with $T_d < t_i - t_{i-1} \leq T_p$, and F terminates within T_d from $t_i, \forall i$.*

In short, the PRW has the ability to periodically execute well-defined functions in known bounded time. Moreover, the PRW allows the definition of a set of fail-safe measures to be triggered in certain situations. For instance, these fail-safe measures may shutdown the system or alert an administrator if the *periodic timely*

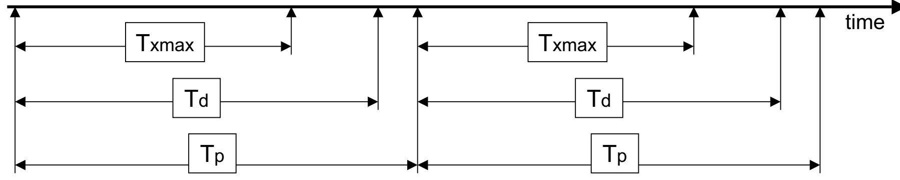

Figure 2: Relationship between the period T_p, the maximum execution time T_d and the worst case execution time T_{Xmax}.

execution service fails to satisfy its specification. These self-checking mechanisms can then be used to prevent the occurrence of resource exhaustion even if the proactive recovery procedure fails to achieve its goal.

A PRW instantiation is defined by a triple $\langle D, \langle F, T_p, T_d \rangle, S \rangle$, such that:

- D represents the set of *data* which is proactively recovered, in all nodes (e.g., private key shares as in CODEX [19], system state as in BFT-PR [6]);
- $\langle F, T_p, T_d \rangle$ represents the *function* F which is periodically triggered with period T_p and timely executed within T_d of each triggering, through the *periodic timely execution* service, in all nodes. F makes operations over the data defined in D;
- S represents the set of *self-checking* mechanisms, which have the goal of guaranteeing a fail-safe behaviour of all the nodes.

The feasibility of a specific PRW instantiation is assessed at design time. The system architect defines, at design time, the function F corresponding to the proactive recovery procedure to be executed, as well as, its required periodicity T_p and execution interval T_d. A PRW instantiation is feasible if T_p is greater than T_d and T_d is greater than the worst-case execution time of F.

2.3.2 Building Replica-Exhaustion-Safe Distributed f Fault/Intrusion-Tolerant Systems

In order to build a replica-exhaustion-safe distributed f fault/intrusion-tolerant system, one has to guarantee that no more than f (accidental or malicious) replica failures occur during system execution. If the system maximum execution time is known, then one may choose a sufficiently large f – by endowing the system with sufficient replicas – so that resource exhaustion never occurs. However, if the system has an unbounded execution time, we have a problem – it is not possible to estimate how many replicas will be needed to avoid resource exhaustion. One possible approach to solve this problem is to use the Proactive Resilience Model – enhance the system with a PRW in order that replicas are periodically and timely rejuvenated. Notice that this approach may even be applied in systems with a

known bound on execution time when there is the need of minimizing the number of used replicas.

We propose a design methodology to build replica-exhaustion-safe distributed f fault/intrusion-tolerant systems, under the Proactive Resilience Model. The methodology has 3 steps.

Definition 2.5. *The design methodology $\mathcal{M}_{exhaustion-safe}$ is defined by the following steps:*

1. *Define the data D to rejuvenate, the rejuvenation procedure F, the required periodicity T_p, the execution interval T_d, and the actions S to be performed if F is not executed with the required periodicity and execution time.*
2. *Build a PRW instantiation $\langle D, \langle F, T_p, T_d \rangle, S \rangle$.*
   - *If not feasible, increase the values of T_p and/or T_d.*
3. *Define the degree f_{safe} of fault-tolerance, such that, the minimum time necessary – $T_{exhaust_{min}}$ – for $f_{safe} + 1$ replica failures to be produced satisfies the condition $T_{exhaust_{min}} > T_p + T_d$.*

In [25] we formally show that a system built using this methodology is replica-exhaustion-safe.

3 A Replica-Exhaustion-Safe State Machine Replication System

3.1 State Machine Replication

A state machine is defined by a set of state variables and a group of commands. The collection of state variables defines the state of the system. Commands are used to perform modifications on the state variables and/or to produce some output (e.g., read the value of a state variable) [22]. Almost every computer program can be modelled as a state machine. In particular, we will focus on client/server applications, which also fit under this model: the server is responsible for maintaining the state and the clients issue commands that modify or read the state. This way of looking at client-server applications facilitates the reasoning on how to make these type of applications fault-tolerant. The simplest form of implementing a client-server application is by deploying a single centralized server which processes all the commands issued by clients. As long as the server does not fail, commands are performed according to the order they are received from clients. But if one considers that failures may happen, then this centralized approach does not work. The server may crash and render the system unavailable or, worst, the server may be compromised by some malicious adversary, which can arbitrarily modify the state.

In order to tolerate these types of failures, one has to replicate the server. The replication degree depends both on the type (e.g., crash, Byzantine) and quantity of the failures to be tolerated. Several protocols have been proposed to implement state machine replication tolerant to crash faults, and some also targeting the Byzantine scenario. Given that our focus in this paper is on Internet services, we will not make any restrictions on the type of faults than can happen – a server may fail arbitrarily, either by crash or by compromise of the state and/or the execution logic. The current state-of-the-art allows one to build client/server applications resilient to a specified number f of arbitrary faults – we call this *f-resilient* systems.

f-resilient systems (with $f \geq 1$) are not necessarily better than systems without fault-tolerance. In fact, given that f-resilient systems have an increased complexity, the performance of the replicated system is typically worse than of the centralized version. The advantage is, off course, the resilience to a certain number of faults. However, the actual resilience of the replicated system depends both on the correlation between replica failures and on the strength of the malicious adversary. On the one hand, if all the replicas use the same operating system and the service implementation is equal in all of them, then, an adversary only needs to discover how to compromise a single replica in order to easily compromise more than f replicas. On the other hand, even if all the replicas operate over different operating systems and use different implementations, a malicious adversary with the ability of triggering attacks in parallel may substantially reduce the time needed to corrupt more than f replicas. For instance, if the adversary is a group of equally powered $f+1$ hackers working in parallel and trying to compromise $f+1$ different replicas, the time necessary to corrupt more than f replicas would correspond to the time necessary to compromise the less vulnerable replica. Moreover, in long-lived systems, even if replicas are attacked in sequence, the probability of more than f being compromised is significant. From this reasoning, we identify three key factors that influence the actual resilience of a f fault-tolerant system:

- Failure correlation:
  - Completely different replicas: different operating system, different service design and implementation;
  - Similar replicas: same operating system and/or same design and/or same implementation;
  - Completely equal replicas: same operating system, same service design and implementation.
- Adversary strength:
  - Replicas are attacked in sequence;
  - Replicas are attacked in parallel.
- Total execution time of the replicated system.

The first two factors determine the time needed to corrupt more than f faults. By assessing if this value is greater or lower than the total execution time of the system, one can determine the resilience of an f fault-tolerant replicated system.

It is extremely important that no more than f faults can occur during system execution. In order to build truly dependable systems, one has to guarantee this bound on the number of faults by construction. With this goal in mind, proactive recovery seems to be a very interesting approach: replicas can be periodically rejuvenated and thus the effects of accidental and/or malicious faults can be removed. However, proactive recovery execution needs some synchrony guarantees in order that rejuvenations are regularly triggered and have a bounded execution time.

In the case of state machine replication, we argue that despite being difficult to guarantee a bounded execution time on the proactive recovery procedure, because it involves time-consuming tasks such as state transfer, one can devise an architecture under which anomalous recovery times can be dependably detected before more than f replicas being compromised. We propose to apply the Proactive Resilience Model (PRM), introduced in Section 2.3, to the state machine replication scenario. Under PRM, the proactive recovery procedure is executed in the context of a synchronous and secure distributed architectural component – the Proactive Recovery Wormhole (PRW). The PRW timely execution service is used to proactively recover replicas, guaranteeing that:

- no more than f replicas are ever corrupted;
- the execution of the distributed state machine is never interrupted.

Our approach is minutely explained in the next section.

3.2 The State Machine Proactive Recovery Wormhole

Previously, we have described the PRW as an abstract component, and in this section, we give an instantiation of the PRW for state machine replication – the State Machine Proactive Recovery Wormhole (SMW). The goal of the SMW is to periodically rejuvenate replicas such that no more than f replicas are ever compromised and thus replica-exhaustion-safety is guaranteed.

The SMW is defined by the triple $\langle D_{SMW}, F_{SMW}, S_{SMW} \rangle$, such that:

- $D_{SMW} = \{$ `OS code`, `SM code`, `SM state` $\}$, where `OS/SM code` is the code of the operating system/state machine and `SM state` is the state of the state machine. These are the three types of data to be periodically refreshed.
- $F_{SMW} = \langle$ `refreshCodeAndState`, $T_p, T_d \rangle$, where the concrete values of T_p and T_d depend on several factors that will be discussed later in the section, and the `refreshCodeAndState` function is presented as Algorithm 1. Each non crashed local SMW $P_i, i \in \{1..n\}$, executes Algorithm 1

Algorithm 1 refreshCodeAndState() for each local SMW $P_i, i \in \{1...n\}$

1: $shutdownOS()$

2: **if** OS code is corrupted **then** {restore operating system code}
3: $restoreOScode()$

4: **if** SM code is corrupted **then** {restore state machine code}
5: $restoreSMcode()$

6: $bootOS()$ {at this point, the OS and the SM can be safely booted because their code is correct}

7: wait until state recovery is finished

at some point of each time period defined by T_p. The precise execution start instant depends on the recovery strategy that will be discussed in Section 3.3.

- $S_{SMW} = \{$switch to a fail-safe state and/or alert an administrator, if the state is not periodically and timely rejuvenated, as specified by T_p and $T_d\}$.

Regarding Algorithm 1, we assume that the state of both operating system and local state machine is stored in volatile Random-Access Memory (RAM). Moreover, the state of the local state machine is periodically saved to stable storage. Also, we assume that the local state machine is automatically started after every boot of the operating system, and that the previous state is loaded from the stable storage.

In Algorithm 1, Line 1 shutdowns the operating system, and consequently stops the execution of the local state machine. Notice that the algorithm continues to execute even after the operating system being shutdown. This happens because the SMW does not depend on the operating system, which can be achieved in practice by implementing each local SMW in a PC board. Line 2 checks if the operating system code is corrupted. To accomplish this task, a digest of the operating system code can be initially stored on some read-only memory, and then assessing if it is correct is only a matter of comparing the digest of the current code with the stored one. In Line 3, the operating system code can be restored from a read-only medium, such as a Read-Only Memory (ROM) or a write-protected hard disk (WPHD), where the write protection can be turned on and off by setting a jumper switch (e.g., Fujitsu MAS3184NP). In Lines 4–5, the state machine code can be checked and restored using similar methods to the ones we used to check and restore the operating system code. Alternatively, both the operating system and the state machine code can be installed on a read-only medium, thus avoiding the execution of Lines 2–5. Line 6 boots the operating system from a clean code and thus brings it to a correct state. The local state machine is also automatically started.

Given that the state of the local state machine may have been compromised before the rejuvenation, it may be necessary to transfer a clean state from remote replicas. In Line 7, we wait until a potential state recovery is finished. A generic state recovery mechanism for fail-stop replicas is described in [22]. This mechanism can be easily generalized to the case when we can have Byzantine failures. State recovery introduces an unbounded delay on the proactive recovery procedure, given that it requires the exchange of information through the payload network. Since the payload network is asynchronous, messages sent through it can take an unbounded time to be delivered. However, one can estimate an upper-bound on the delivery time which will be satisfied with high probability in normal conditions.

In the worst-case scenario, i.e., when the code of both the operating and the local state machine is corrupted, Algorithm 1 executes a total of 7 operations. The execution time of these operations can be upper-bounded in the following manner.

- $shutdownOS()$: Typically, an operating system can be shutdown though an hardware interrupt. This operation has predictable execution time and thus one can define an upper-bound $T_{shutdown}$.
- check OS/SM code correctness: This can be implemented through a bitwise comparison between the digest of the current OS/SM code with a digest initially stored in a read-only medium. An upper-bound $T_{checkcode}$ on the execution time of this operation can be defined based on the maximum time necessary to copy the digest from the medium and to make a simple bitwise comparison.
- $restoreOScode()$ and $restoreSMcode()$: Restoring the OS/SM code corresponds to a copy from a read-only medium. So, an upper-bound $T_{restorecode}$ can be defined based on the maximum time necessary to copy the code from the medium.
- $bootOS()$: Booting an operating system can take some time, but given that we are always booting the same OS (because no application gets installed between boots), it is possible to estimate an upper-bound T_{boot} on the boot time.
- wait until recovery is finished: One can estimate an upper-bound $T_{transfer}$ on the transfer time which will be satisfied with high probability in normal conditions.

So, one can define an upper-bound $T_{localexec}$ on the execution time of Algorithm 1, such that $T_{localexec} = T_{shutdown} + 2T_{checkcode} + 2T_{restorecode} + T_{boot} + T_{transfer}$. This upper-bound will be used in the next section as an input to calculate the values of T_p and T_d. The fail-safe mechanisms ensure that if the recovery procedure does not terminate within its deadline (e.g., due to the violation of an estimated upper-bound), all the replicas switch to a fail-safe state (e.g., shutdown) and/or an alert is sent to an administrator. Notice that the fail-safe mechanisms

will only be triggered under abnormal conditions, when the system is in danger of becoming compromised.

3.3 Strategies to Achieve Replica-Exhaustion-Safety

We now discuss the recovery strategy to be applied in order that no more than f replicas are ever corrupted, and the execution of the distributed state machine is never interrupted. Consider that there exists a function $T_{exhaust}(x)$ which returns the minimum time needed to compromise x replicas.

If all replicas are periodically rejuvenated within a period T_p and the rejuvenation execution time is bounded by T_d, then one can guarantee a maximum of f faulty replicas if $T_p + T_d < T_{exhaust}(f+1)$. A straightforward solution to achieve this objective would be to rejuvenate all the replicas at once: the replicas would be simultaneously stopped in a consistent state, rejuvenated, and restarted again. Given that no progress would occur during rejuvenation, only the previously compromised replicas would have to restore their state. The problem with this solution is that the distributed state machine would be unavailable during the rejuvenation, which is contrary to one of our goals. However, in scenarios where the interruption of the service is not a problem, this solution has the advantage of minimizing the number of state transfers, given that only compromised states have to be restored.

In order to avoid service interruption, the number k of replicas simultaneously recovered should be such that $k \leq f$. If k is greater than f, than the state machine may not continue its operation during a recovery. Moreover, we have to guarantee that the maximum number k' of faults that can occur between rejuvenations is such that $k + k' \leq f$. In the worst case scenario, k' replicas are compromised when a different set of k replicas are recovering. The reader, however, should notice that k and k' may be chosen at will, according to the requirements of the service offered by the distributed state machine. On the one hand, if k has a low value then few replicas recover simultaneously and more faults are allowed to happen between rejuvenations. On the other hand, if k has an high value, then more replicas will be recovered at the same time, but less faults will be tolerated between rejuvenations.

Each recovering replica executes the code presented in Algorithm 1. Replicas are recovered in groups of at most k elements, by some specified order: for instance, replicas $P_1, ..., P_k$ are recovered first, then replicas $P_{k+1}, ..., P_{2k}$ follow, and so on. A total of $\lceil n/k \rceil$ replica groups are rejuvenated in sequence. Figure 3 illustrates the rejuvenation process. The SMW coordinates the rejuvenation process, triggering the rejuvenation of replica groups one after the other. The maximum execution time of the rejuvenation process, i.e., the maximum time interval between the first group rejuvenation start instant and the last group rejuvenation termination instant, is upper-bounded by $T_{exec} = \lceil n/k \rceil T_{localexec}$.

Therefore, by applying the methodology $\mathcal{M}_{exhaustion_safe}$, if $T_d \geq T_{exec}$ and $T_p > T_d$, then the system is replica-exhaustion-safe if we choose k' (with $k + k' \leq f$), such that, $T_p + T_d < T_{exhaust}(k'+1)$.

Given that we need to tolerate at least one faulty replica between rejuvena-

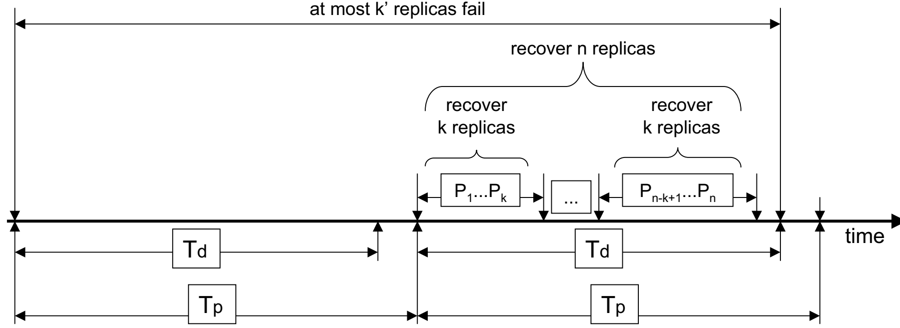

Figure 3: Relationship between the rejuvenation period T_p, the rejuvenation maximum execution time T_d, k and k'.

tions[1], k' should be greater than zero. So, at least one replica will be recovered per rejuvenation (i.e., $k \geq 1$) and thus $f \geq 2$. Therefore, a Byzantine fault-tolerant state machine (where $n \geq 3f + 1$) should apparently have a minimum of 7 replicas in order to satisfy availability and replica-exhaustion-safety. However, albeit k' faults may be of Byzantine nature, the k faults provoked by the rejuvenation process are fail-silent. So, we need in fact $n \geq 3k' + 2k + 1$ replicas, and thus a minimum of 6 replicas.

4 Related Work

Proactive recovery [20] has been used in different scenarios with the goal of restricting the assumption on a bounded number of faults to a small interval.

4.1 Secret Sharing

Secret sharing schemes [1, 23] protect the secrecy and integrity of secrets by distributing them over different locations. A secret sharing scheme transforms a secret s into n shares $s_1, s_2, ..., s_n$ which are distributed to n share-holders. In this way, the adversary has to attack multiple share-holders in order to learn or to destroy the secret. For instance, in a $(k + 1, n)$-threshold scheme, an adversary needs to compromise more than k share-holders in order to learn the secret, and corrupt at least $n - k$ shares in order to destroy the same secret.

In many applications, a secret s may be required to be held in a secret-sharing manner by n share-holders for a long time. If at most k share-holders are corrupted throughout the entire lifetime of the secret, any $(k + 1, n)$-threshold scheme can be used. In certain environments, however, gradual break-ins into a subset of locations over a long period of time may be feasible for the adversary. If more than

[1]We could assume no faults between rejuvenations, but then we would be assuming that the adversary would be unable to compromise any replica.

k share-holders are corrupted, s may be stolen. An obvious defense is to periodically refresh s, but this is not possible when s corresponds to inherently long-lived information (e.g., cryptographic root keys, legal documents).

Thus, what is actually required to protect the secrecy of the information is to be able to periodically renew the shares without changing the secret. Proactive secret sharing (PSS) was introduced in [13] in this context. In PSS, the lifetime of a secret is divided into multiple periods and shares are renewed periodically. In this way, corrupted shares will not accumulate over the entire lifetime of the secret since they are checked and corrected at the end of the period during which they have occurred. A $(k+1, n)$ proactive threshold scheme guarantees that the secret is not disclosed and can be recovered as long as at most k share-holders are corrupted during each period, while every share-holder may be corrupted multiple times in some periods.

Different approaches to proactive secret sharing have been suggested, both under the synchronous and the asynchronous model. Synchronous approaches [13, 12, 11] work correctly in a synchronous environment, which is not the environment tackled in this paper. Asynchronous proactive secret sharing [28, 3, 27, 19], on the other hand, fail to guarantee a bound on the rejuvenation period: a malicious adversary may slow the system down (e.g., by compromising the clock behaviour).

4.2 State Machine Replication

Castro and Liskov were the first ones to propose the combination of asynchronous state machine replication with proactive recovery [6]. They propose BFT-PR – a Byzantine fault-tolerant, state machine replication algorithm, which uses proactive recovery. BFT-PR can tolerate any number of faults provided fewer than one third of the replicas become faulty within a window of vulnerability.

BFT-PR works mainly under the asynchronous model, but the proactive recovery mechanism makes some extra assumptions: secure cryptography, read-only memory, watchdog timers and eventual timely delivery of messages. Secure cryptography means that a replica can sign and decrypt messages without exposing its private key. Read-only memory is used both to store the public keys for other replicas and to store the recovery monitor that executes the (proactive) recovery procedure. Watchdog timers are used to periodically interrupt processing and hand control to the recovery monitor. Therefore, each replica needs to be equipped with a secure cryptographic coprocessor, a watchdog timer and a recovery monitor. It is also assumed that there is some unknown point in the execution after which either all messages are delivered within some constant time Δ or all non-faulty clients have received replies to their requests.

If these assumptions are satisfied, then BFT-PR works correctly. Namely, authors point out that Δ is a constant that depends on the *timeout* values used by the algorithm and that an appropriate choice of Δ allows recoveries at a fixed rate. This suggests that the length T_v of the window of vulnerability can have a known bounded value in normal conditions.

However, given that BFT-PR targets malicious environments (e.g., Internet), the bound on T_v should either be guaranteed under an attack, or it should be possible to timely detect any increase on the window of vulnerability and then take the necessary actions in order to avoid the compromise of the system. The problem is that BFT-PR does not provide adequate mechanisms to achieve any of these goals. Our approach, on the other hand, guarantees precisely this behaviour: in normal conditions, the rejuvenation is periodically and timely executed, and if some abnormal situation occurs, the system switches to a fail-safe state and/or alerts an administrator, before being compromised.

5 Conclusions and Future Work

One of the current main challenges is how to build and deploy highly available services on the Internet. The traditional approach, and the one which seems more appropriate, is based on replication. But replication has costs, namely on how to guarantee a correct coordination between the replicas. These costs are even higher on the Internet, given that its unpredictability and insecurity forces the coordination protocols to be Byzantine fault-tolerant and immune to timing failures. Several agreement and replication techniques of this type were already described in the past, but all of them make the dangerous assumption that the number of faulty replicas is bounded by a known value during an unbounded execution time interval.

In this paper, we described a resilient f fault/intrusion-tolerant state machine replication system, which guarantees that no more than f faults ever occur. The system is asynchronous in its most part, using a synchronous oracle – the State Machine Proactive Recovery Wormhole – to periodically remove the effects of faults/attacks from the replicas. We performed a quantitative assessment of the level of redundancy required to achieve resilient state machine replication, i.e., simultaneously securing availability and replica-exhaustion-safety. We see that 6 replicas are required for tolerating one Byzantine failure, versus the 4 replicas required in sheer algorithmic terms, believed until now sufficient.

As future work, we plan to implement an experimental prototype of the proposed state machine replication system.

References

[1] G. R. Blakley. Safeguarding cryptographic keys. In *Proceedings of the National Computer Conference*, volume 48 of *AFIPS*, pages 313–317. 1979.

[2] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840, Oct. 1985.

[3] C. Cachin, K. Kursawe, A. Lysyanskaya, and R. Strobl. Asynchronous verifiable secret sharing and proactive cryptosystems. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 88–97. ACM Press, 2002.

[4] C. Cachin, K. Kursawe, and V. Shoup. Random oracles in Contanstinople: Practical asynchronous Byzantine agreement using cryptography. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing*, pages 123–132, July 2000.

[5] R. Canetti and T. Rabin. Fast asynchronous Byzantine agreement with optimal resilience. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*, pages 42–51, 1993.

[6] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, Nov. 2002.

[7] M. Correia, N. F. Neves, and P. Veríssimo. How to tolerate half less one Byzantine nodes in practical distributed systems. In *Proceedings of the 23rd IEEE Symposium on Reliable Distributed Systems*, Oct. 2004.

[8] F. Cristian and C. Fetzer. The timed asynchronous system model. In *Proceedings of the 28th IEEE International Symposium on Fault-Tolerant Computing*, pages 140–149, 1998.

[9] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper. Muteness failure detectors: Specification and implementation. In *EDCC-3: Proceedings of the Third European Dependable Computing Conference on Dependable Computing*, pages 71–87, London, UK, 1999. Springer-Verlag.

[10] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, Apr. 1985.

[11] J. A. Garay, R. Gennaro, C. Jutla, and T. Rabin. Secure distributed storage and retrieval. *Theor. Comput. Sci.*, 243(1-2):363–389, 2000.

[12] A. Herzberg, M. Jakobsson, S. Jarecki, H. Krawczyk, and M. Yung. Proactive public key and signature systems. In *Proceedings of the 4th ACM Conference on Computer and Communications Security*, pages 100–110. ACM Press, 1997.

[13] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung. Proactive secret sharing or: How to cope with perpetual leakage. In *Proceedings of the 15th Annual International Cryptology Conference on Advances in Cryptology*, pages 339–352. Springer-Verlag, 1995.

[14] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[15] N. Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.

[16] D. Malkhi and M. Reiter. Byzantine quorum systems. In *STOC '97: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 569–578, New York, NY, USA, 1997. ACM Press.

[17] D. Malkhi and M. Reiter. Unreliable intrusion detection in distributed computations. In *Proceedings of the 10th Computer Security Foundations Workshop*, pages 116–124, June 1997.

[18] D. Malkhi and M. Reiter. An architecture for survivable coordination in large distributed systems. *IEEE Transactions on Knowledge and Data Engineering*, 12(2):187–202, 2000.

[19] M. A. Marsh and F. B. Schneider. CODEX: A robust and secure secret distribution system. *IEEE Transactions on Dependable and Secure Computing*, 1(1):34–47, January–March 2004.

[20] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks (extended abstract). In *Proceedings of the tenth annual ACM symposium on Principles of distributed computing*, pages 51–59. ACM Press, 1991.

[21] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, Apr. 1980.

[22] F. B. Schneider. Implementing faul-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, Dec. 1990.

[23] A. Shamir. How to share a secret. *Commun. ACM*, 22(11):612–613, 1979.

[24] P. Sousa, N. F. Neves, and P. Veríssimo. How resilient are distributed f fault/intrusion-tolerant systems? In *Proceedings of the Int. Conference on Dependable Systems and Networks*, pages 98–107, June 2005.

[25] P. Sousa, N. F. Neves, and P. Veríssimo. Proactive resilience through architectural hybridization. DI/FCUL TR 05–8, Department of Informatics, University of Lisbon, May 2005. http://www.di.fc.ul.pt/biblioteca/techreports/05-8.pdf. Submitted for publication.

[26] P. Veríssimo. Uncertainty and predictability: Can they be reconciled? In *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*, pages 108–113. Springer-Verlag, 2003.

[27] L. Zhou, F. Schneider, and R. van Renesse. COCA: A secure distributed on-line certification authority. *ACM Transactions on Computer Systems*, 20(4):329–368, Nov. 2002.

[28] L. Zhou, F. B. Schneider, and R. van Renesse. Proactive secret sharing in asynchronous systems. Technical Report TR 2002-1877, Cornell University, Ithaca, New York, Oct. 2002.